第二章
树：树与二叉树、字典树

在一片茂密的森林中，每一棵树都从根部向上生长，分出枝干，再蔓延出细小的叶片，形成层次分明的结构。这种自然的层级之美，正是计算机科学中树形结构的灵感来源——无论是家族谱系中代代相传的血脉，还是文件系统中目录与子目录的嵌套关系，树都能以简洁的规则描述复杂的层次逻辑。

在编程的世界里，树是一种非线性数据结构，它摆脱了线性结构的单一顺序，用分支和层级为数据赋予更丰富的表达力。从数据库索引的快速检索，到网络路由的高效寻址；从决策树的智能分类，到语法树的代码解析，树的身影无处不在。本章将带您从基础出发，深入理解树的数学定义与核心术语，掌握其存储方法与遍历技巧，并通过经典问题剖析树在算法中的巧妙应用。

2.1 树的基本概念

本节学习目标

- 理解树的数学定义与基本性质，能够区分树与图的区别。
- 掌握树的基本术语（如根、叶、深度、高度等）及其实际意义。
- 识别特殊树类型（链、菊花、二叉树）的结构特点。
- 掌握邻接表存储树的方法，并能用代码实现树的遍历。

2.1.1 问题引入：家谱关系建模

问题描述

某家族需要构建家谱图，要求表示成员间的父子关系，并支持快速查询祖先、后代及兄弟关系。如何用数据结构高效表示这种层级关系？

具体实例

假设家族成员关系如下：

（1）A 是 B 和 C 的父亲

（2）B 是 D 和 E 的父亲

（3）C 是 F 的父亲

2.1.2 知识核心

树的定义

树是满足以下任一条件的连通无向图：

- 有 n 个结点和 n-1 条边，且无环。
- 任意两结点间有且仅有一条简单路径。
- 所有边均为桥（删除任意边会导致图不连通）。

如图2.1.1，一棵包含 6 个结点的树，边连接方式为层级结构。

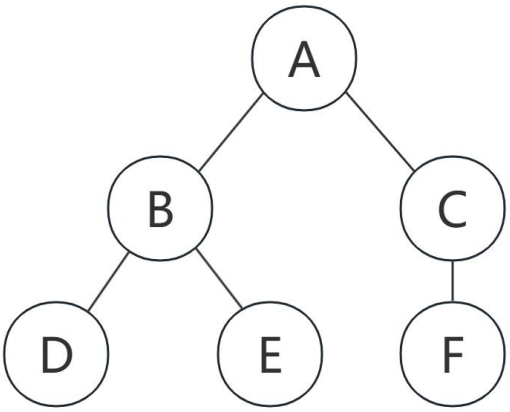


图2.1.1 树结构示意图

树的基本概念

◆ **父结点（parent node）**

在树结构中，若结点 A 直接连接到结点 B 的上层，则称 A 是 B 的父结点。除根结点外，每个结点有且仅有一个父结点。根结点没有父结点。

◆ **祖先（ancestor）**

若存在从结点 B 到结点 A 的路径，则 A 是 B 的祖先。根结点的祖先集合为空。

◆ **子结点（child node）**

若结点 B 是结点 A 的直接下层结点，则 B 是 A 的子结点。一个结点可以有零个或多个子结点，具体数量由树的类型决定。

◆ **结点的深度（depth）**

从根结点到该结点的路径上经过的边的数量。

◆ **树的高度（height）**

从某结点到其最远叶子结点的最长路径上的边数。

◆ **兄弟（sibling）**

具有相同父结点的结点互为兄弟结点。

◆ **后代（descendant）**

若存在从结点 A 到结点 B 的路径（通过连续向下访问子结点），则 B 是 A 的后代。

◆ **子树（subtree）**

以树中某结点为根结点，包含其所有后代结点及连接的边构成的子结构。

◆ **森林（forest）**

由零个或多个互不相交的树组成的集合。

◆ **无根树的叶结点（leaf node）**

在无根树中，度数为 1 的结点称为叶结点。

◆ **有根树的叶结点（leaf node）**

在有根树中，没有子结点的结点称为叶结点。

特殊的树

（1）链（chain/path graph）：链是一种退化的树结构，所有结点按线性顺序排列，每个结点（除首尾结点外）仅有一个父结点和一个子结点。

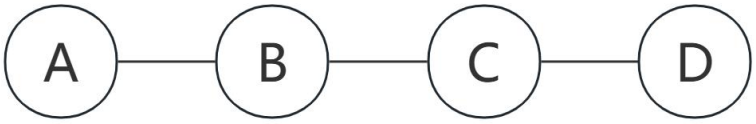


图2.1.2 链

（2）菊花/星星（star）：由一个中心根结点和多个直接连接的叶结点组成，无中间层级的树结构。

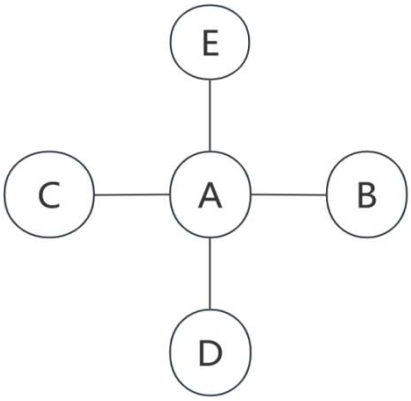


图2.1.3 菊花/星星

（3）有根二叉树（rooted binary tree）：有根二叉树是一种每个结点最多有两个子结点的有根树。

2.1.3 利用邻接表存储树

理解问题

邻接表由顶点数组（或哈希表）和链表（或动态数组）组成。每个顶点对应一个独立的链表，链表中存储与该顶点直接相连的所有相邻顶点。

已知要利用邻接表存储树，我们需要为每个结点开辟一个线性列表，记录所有与之相连的结点。

示例展示

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int N = 100010;
vector<int> adj[N];

int main() {
    int n, m; /
    cin >> n >> m;
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    return 0;
}
```

图2.1.4　利用邻接表存储树

2.1.4 利用DFS遍历树

概念介绍

DFS（Depth-First Search，深度优先搜索）是一种用于遍历或搜索树、图等数据结构的算法。其核心思想是尽可能深地探索分支路径，直到无法继续前进，再回溯到上一个分叉点尝试其他路径。

示例展示

```cpp
void dfs (int cnt, int f) {
    for (int i : adj[cnt]) {
        if (i ≠ f) {
            dfs(i, cnt);
        }
    }
}
```

图2.1.4　利用DFS遍历树

思考辨析

如果将示例程序中的 if(i != f) 语句删去会发生什么？

删除示例程序中的 if (i != f) 会导致无限递归和栈溢出。if (i != f) 是确保 DFS 正确遍历的关键条件，删除后程序将无法正常运行。

◆ **原因：**

1. **防止回溯父节点：**该条件的作用是避免当前节点 i 重新访问其父节点 f。在树或图的遍历中，父节点已被处理过，无需重复访问。

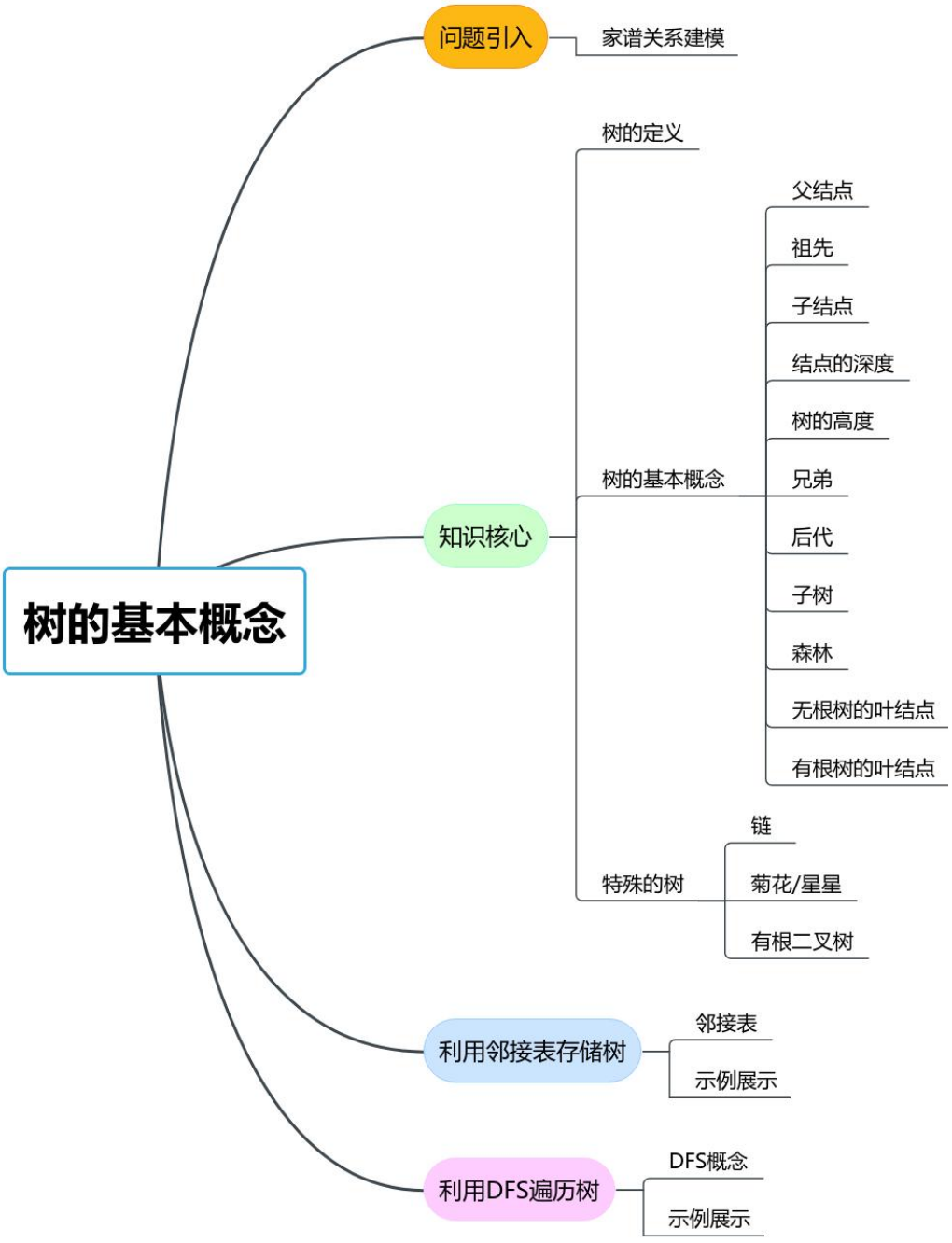
2. **避免循环：**若删除此条件，当遍历到父节点时，程序会递归调用 dfs(f, cnt)，而 f 又会再次访问当前节点 cnt，形成 cnt ↔ f 的无限循环。

3. **栈溢出：**递归深度无限制增加，最终因函数调用栈耗尽而崩溃。

◆ **示例：**

假设节点 1 和 2 相连。调用 dfs(1, -1) 会访问 2，接着调用 dfs(2, 1)。若未跳过父节点，2 会再次访问 1，触发 dfs(1, 2)，1 又访问 2……无限循环。

知识图谱


树的基本概念
问题引入
家谱关系建模
知识核心
树的定义
树的基本概念
父结点
祖先
子结点
结点的深度
树的高度
兄弟
后代
子树
森林
无根树的叶结点
有根树的叶结点
特殊的树
链
菊花/星星
有根二叉树
利用邻接表存储树
邻接表
示例展示
利用DFS遍历树
DFS概念
示例展示

练习提升

一、判断题

1. 树中根节点没有父节点，其他节点有且仅有一个父节点。（　）

2. 叶子节点是指没有子节点的节点。（　）

3. 树中任意两个节点之间的边构成一条唯一的路径。（　）

4. 节点的“度”是指该节点的兄弟节点数量。（　）

二、填空题

1. 树中节点的子节点数量称为该节点的_____。

2. 树中从根节点到某一节点经过的边数称为该节点的_____。

3. 根节点的直接子节点称为其_____，根节点是这些子节点的_____。

4. 多个互不相交的树构成的集合称为_____。

三、选择题

1. 以下关于“树的高度”描述正确的是：

   A) 根节点的高度为0

   B) 树的高度等于最深叶子节点的深度

   C) 树的高度等于节点总数

   D) 树的高度等于边数

2. 节点A的父节点的父节点是节点B，则节点B是节点A的：

   A) 兄弟节点

   B) 祖先节点

   C) 子节点

   D) 后代节点

3. 以下属于树的基本术语的是：

   A) 链表

B) 叶子

C) 哈希表

D) 栈

4. 树中同一父节点的子节点互为：

A) 祖先

B) 后代

C) 兄弟

D) 根

四、术语匹配题

将左侧术语与右侧定义连线：

1. 根节点	A. 树中节点的最大深度
2. 森林	B. 没有父节点的节点
3. 路径	C. 由边连接的节点序列
4. 树的高度	D. 多个互不相交的树

五、简答题

1. 定义解释

什么是树的“子树”？

2. 术语对比

“节点的深度”和“节点的高度”有何区别？

2.2 二叉树——倒置的树

本节学习目标

◆ 掌握二叉树的定义，包括空树和非空树的组成（根结点、左子树、右子树）。

◆ 熟悉关键术语：叶子结点、内部结点、结点的度、深度、高度，并能够通过示意图分析二叉树结构。

◆ 通过递归实现前序、中序、后序遍历，并能够根据表达式树写出对应的遍历结果。

2.2.1 二叉树的定义

形式化定义

二叉树（Binary Tree）是一种满足以下条件的树形数据结构：

（1）**空树**：不包含任何结点的二叉树。

（2）**非空树**：由以下三部分组成：

① **根结点（Root）**：唯一没有父结点的结点。

② **左子树（Left Subtree）**：根结点的左子结点及其后代构成的二叉树。

③ **右子树（Right Subtree）**：根结点的右子结点及其后代构成的二叉树。

每个结点最多有 两个子结点，分别称为 **左子结点（Left Child）**和 **右子结点（Right Child）**。子结点的顺序严格区分，左子结点 ≠ 右子结点，不可交换位置。

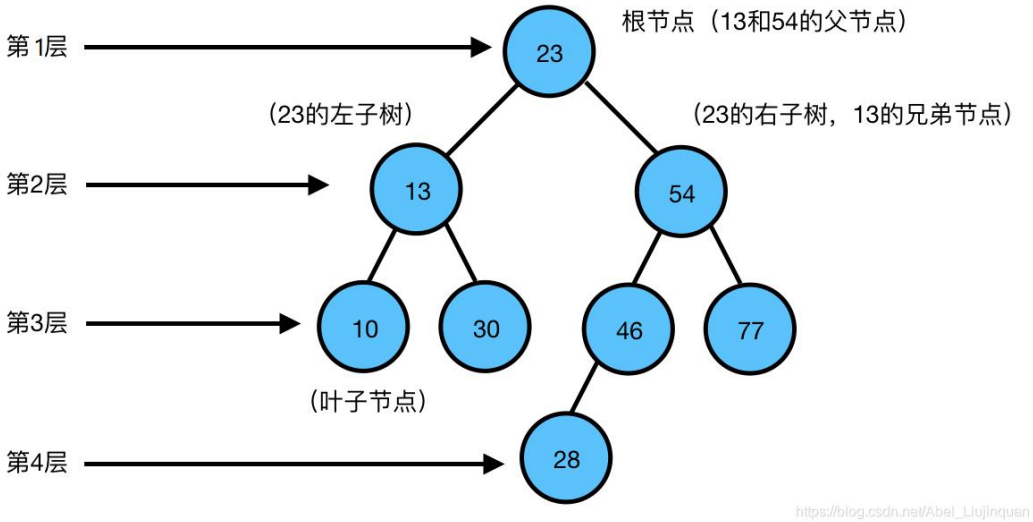


图2.2.1 二叉树结构示意图

关键术语

- **叶子结点（Leaf Node）**

没有子结点的结点（度为0）。

- **内部结点（Internal Node）**

至少有一个子结点的结点（度≥1）。

- **结点的度（Degree）**

一个结点拥有的子结点数量（二叉树中度为0、1或2）。

- **深度（Depth）**

根结点到该结点的路径长度（根结点深度为0或1，依定义约定）。

- **高度（Height）**

结点到叶子结点的最长路径长度（叶子结点高度为0或1，依定义约定）。

2.2.2 二叉树的性质

层、深度与结点数的关系

性质	公式	说明
第 i 层最多结点数	2^(i-1)	几何级数增长
深度 k 的二叉树最大结点数	2^k-1	满二叉树时成立

图2.2.2 层、深度与结点数关系表

结点、度关系原理

设叶子结点数为 n0 ，度为2的结点数为 n2

恒等式：n0=n2+1

证明思路：总边数 B=n1+2n2=n0+n1+n2−1

拓展提升：满二叉树

定义：深度为 k 的二叉树中，所有非叶子结点均有 两个子结点，且所有叶子结点均位于 同一层（第 k层）。

性质：

① 结点总数：2k-1（若根结点深度为1）

② 叶子结点数：2k-1

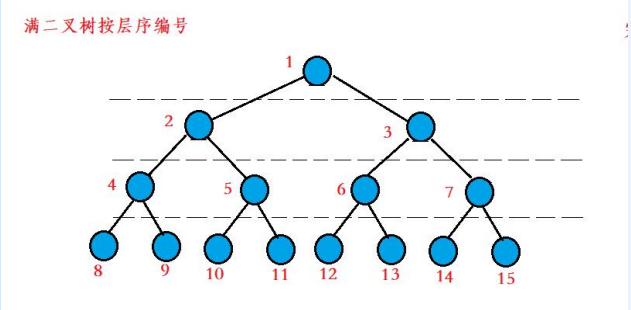


图2.2.3 满二叉树结构示意图

2.2.3 二叉树的存储

结构体数组法

结构体数组法是一种通过数组来静态存储二叉树结点及其子结点关系的实现方式。其核心思想是：

① 用数组下标表示结点的唯一标识（编号）。

② 通过两个独立数组（lchild[] 和 rchild[]）分别记录每个结点的左子结点和右子结点的下标。时也是第一个从栈中弹出的（从一摞盘子中拿出一个盘子，都是从最顶上第一个开始拿）。

```
1 int lchild[N], rchild[N]; // 下标表示结点编号
```

图2.2.4 结构体数组法代码示例

完全二叉树的数组存储

下标 i 的左子为 2*i ，右子为 2*i+1

```
C++
    1   #define ls(X) (x << 1)
    2   #define rs(X) (ls(X) + 1)
```

图2.2.5 完全二叉树的数组存储代码示例

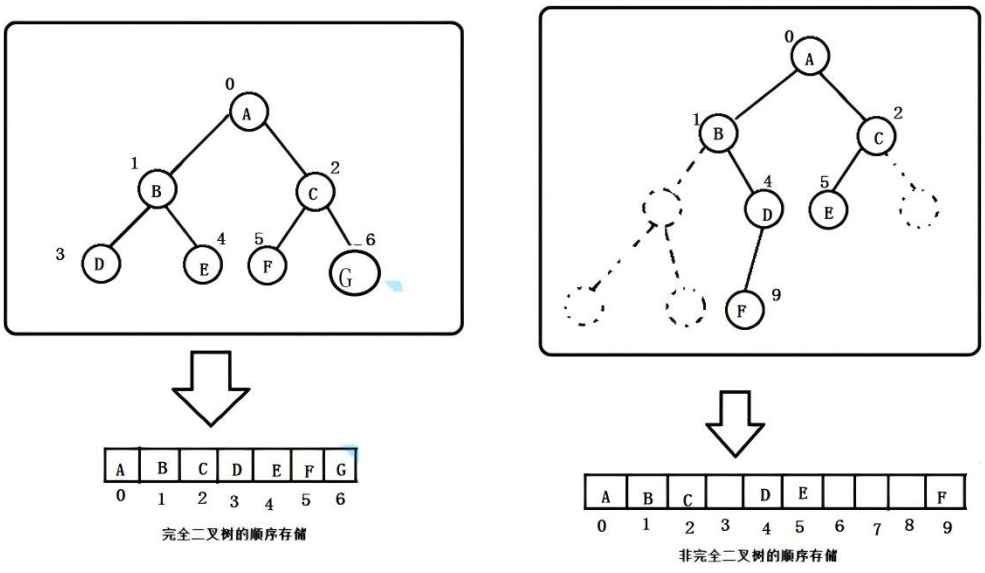


图2.2.6 数组存储与树形结构对应关系图

2.2.4 二叉树的遍历

递归遍历

- **前序遍历（Pre-order Traversal）**：根节点 → 左子树 → 右子树
- **中序遍历（In-order Traversal）**：左子树 → 根节点 → 右子树
- **后序遍历（Post-order Traversal）**：左子树 → 右子树 → 根节点

```
1  void preorder(int u) {
2      if (u == -1) return; // -1 表示空结点
3      cout << u << " ";
4      preorder(lchild[u]);
5      preorder(rchild[u]);
6  }
```

图2.2.7 递归遍历代码示例

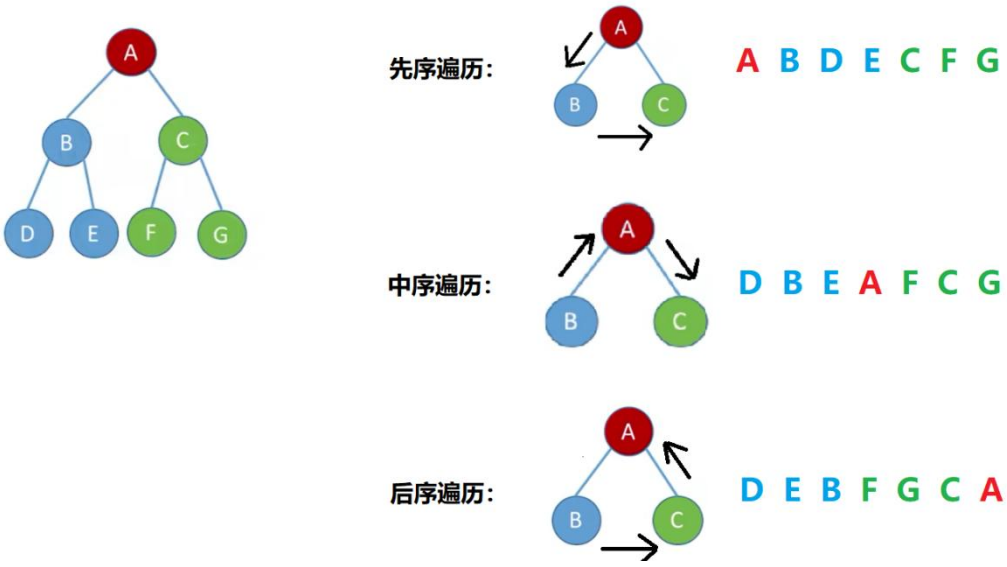


图2.2.8 三种遍历示意图

层次遍历

层次遍历（Level Order Traversal），又称**广度优先遍历（BFS）**，是一种按树的层级逐层访问节点的算法。与递归实现的深度优先遍历（前序/中序/后序）不同，层次遍历使用队列辅助实现，**确保先访问离根节点最近的节点**。

• 核心思想

（1）**按层处理**：从根节点开始，依次访问第1层、第2层……直到最后一层。

（2）**队列辅助**：利用队列的先进先出（FIFO）特性，确保节点按层级顺序被访问。

• 实现步骤

（1）**初始化队列**：将根节点加入队列。

（2）**循环处理队列**：

节点出队 → 访问该节点 → 将子节点（先左后右）入队。

重复直到队列为空。

```C++
queue<int> q;
q.push(root);
while (!q.empty()) {
    int u = q.front(); q.pop();
    cout << u << " ";
    if (lchild[u] != -1) q.push(lchild[u]);
    if (rchild[u] != -1) q.push(rchild[u]);
}
```

图2.2.9 层次遍历代码示例

遍历应用实例：表达式树

表达式树（Expression Tree）是一种用树形结构表示数学表达式的二叉树，能直观反映运算优先级和结合性，。以下是其核心要点：

1. 结构特点

叶子节点：存储操作数（数字或变量）。

内部节点：存储运算符（如 +, -, *, /）。

子树规则：每个运算符节点的左右子树分别表示其操作数。

2. 示例解析

以表达式 3 + 4 * 5 为例：

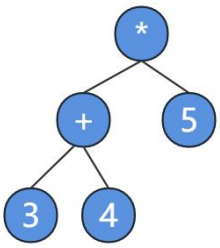


图2.2.10 表达式树示意图

前序对应前缀表达式，中序对应中缀表达式，后序对应后缀表达式。

遍历结果:

前序: * + 3 4 5　　→ 波兰表达式

中序: 3 + 4 * 5　　→ 中缀表达式（需加括号）

后序: 3 4 + 5 *　　→ 逆波兰表达式

思考辨析

已知前序和中序序列，能否唯一确定二叉树？

已知前序遍历和中序遍历序列时，**在节点值唯一的前提下，可以唯一确定一棵二叉树**，这是二叉树遍历序列的重要性质。

◆ **示例演示：**

假设前序序列为 1, 2, 4, 5, 3，中序序列为 4, 2, 5, 1, 3：

1. 前序首元素 1 是根节点。

2. 中序中 1 左侧 4,2,5 是左子树，右侧 3 是右子树。

3. 左子树有3个节点，因此前序中 2,4,5 是左子树的前序序列。

4. 对左子树递归：前序 2 是左子树根，中序 4,2,5 中 2 左侧 4 是左子树，右侧 5 是右子树。

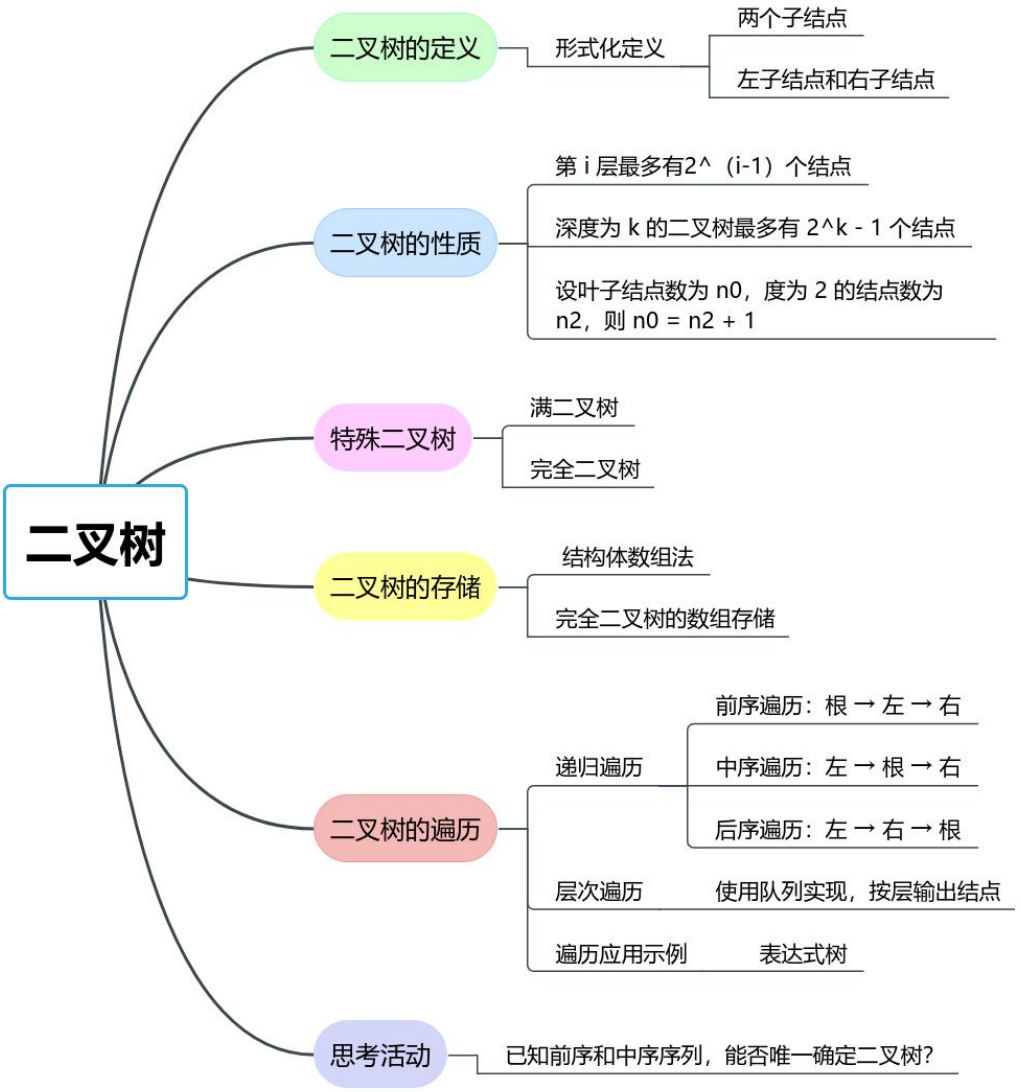
5. 最终重建的树结构唯一。

◆ **特殊情况：**

① 节点值重复：若存在重复值（如两个节点值均为 2），可能导致不同树结构产生相同的前序和中序序列。

② 单节点或空树：此时序列唯一，树结构显然唯一。

知识图谱



练习提升

1. 根据定义，找出图 2.2.11 中的二叉树。

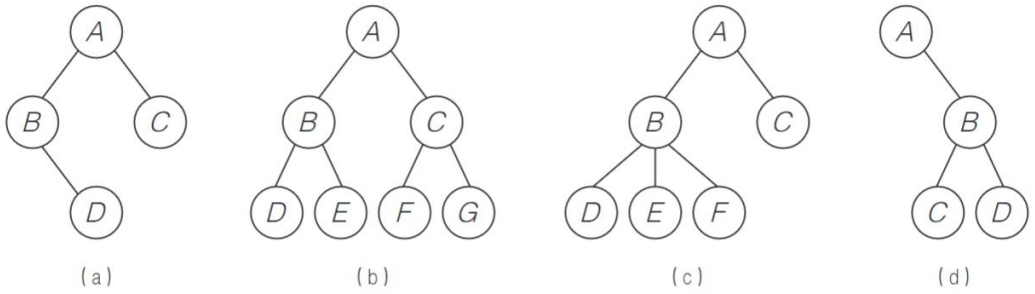


图2.2.11 找出符合条件的二叉树

2. 某二叉树有 3 个节点，两位同学就这棵二叉树的形态产生了争执，一位同学认为该二叉树的形态如图 2.2.12（a），另一位同学则认为该二叉树的形态如图 2.2.12（b）。你赞同哪位同学的看法？还有其他的形态吗？

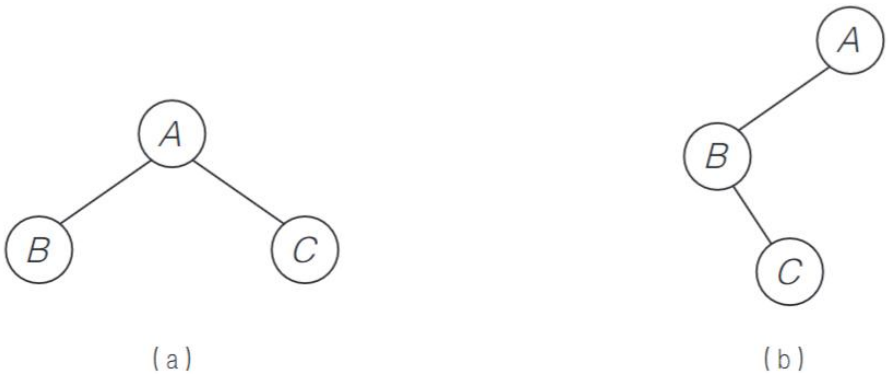


图2.2.12 3个节点的二叉树形态

2.3 树的使用案例——字典树

本节学习目标

- 理解字典树的基本概念与核心特征。
- 掌握字典树的基本操作，并能够解决字典树相关问题。
- 理解字典树的性质，并能够解释其与二叉树对比的优势。
- 识别适合字典树解决的问题场景，独立设计基于字典树的解决方案。
- 能够通过字典树的应用，提升计算机思维和问题解决能力。

2.3.1 字典树的定义

字典树（Tire树）

在计算机的世界里，当需要处理大量字符串时，高效的数据结构就显得尤为重要。今天，我们将一起认识一种专门用于存储和检索字符串的数据结构——**字典树**，也叫**Trie树**。

字典树是一种利用边来存储字符，从而实现多个字符串存储的数据结构。你可以把它想象成一棵**多叉树**，树的每一条路径，都代表着一个字符串。在字典树里，每个节点本身并不存储字符，而是通过节点之间的边所代表的字符，串联构成字符串。（如图2.3.1中a、b等字符存储在节点之间的边中）

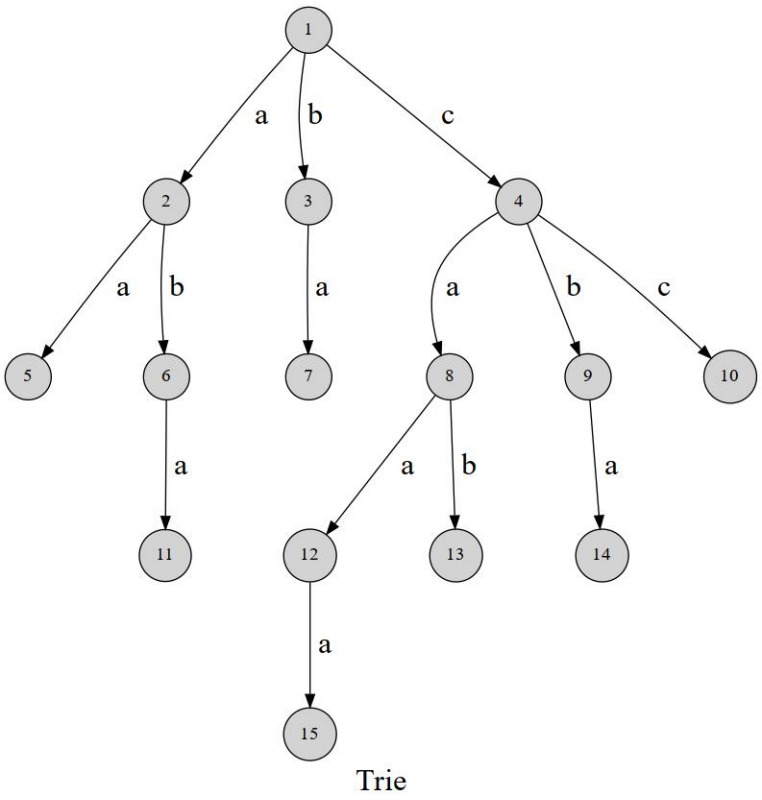


图2.3.1 字典树的结构示意图

具体实例

举个例子，当我们要在字典树中存储“cat”“car”“cart”这几个单词时，从树的根节点出发，第一条边代表字母“c”，通过“c”边到达的节点，再延伸出代表“a”的边，沿着“a”边继续前进，“t”边和“r”边分别对应“cat”和“car”。对于“cart”，则是在“car”路径基础上，从代表“r”的节点继续延伸出“t”边。

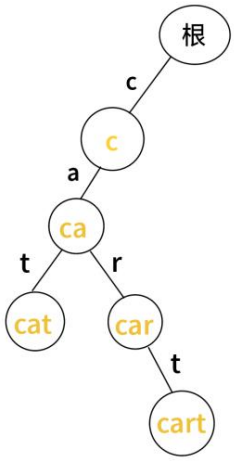


图2.3.2 字典树实例

由以上的图例我们可以看出，字典树有两个**核心特征：多叉树结构、节点不存储字符**。大家也可以发动脑筋进行续写哦！或者用自己喜欢的单词填补右边的字典树！

核心特征1：多叉树结构

字典树与二叉树不同，它的每个节点可以有多个子节点。每个节点的子节点数量，由可能出现的字符数量决定，这些可能出现的字符集合，我们称为“字符集”。在常见的英文字符处理中，字符集大小通常为 26（英文字母共26个）。字典树的每条路径，都唯一对应一个字符串，通过从根节点出发，沿着边所代表的字符顺序拼接，就能得到完整的字符串。

核心特征2：节点不存储字符（路径构成字符串）

在字典树中，节点主要用于引导路径走向，并不存储字符。这种设计让字典树在存储大量具有相同前缀的字符串时，能够节省大量空间。比如，存储“program”“programmer”“programming”时，这些单词共享“program”前缀，在字典树中只需要存储一次该前缀路径，大大减少了存储空间。

字典树与二叉树的对比

字典树与二叉树虽都是树，但二者有很大的不同：

- **子节点数量不固定**

二叉树每个节点最多有两个子节点，而字典树每个节点的子节点数量，取决于字符集的大小。以英文字母为例，字典树每个节点最多可能有26个子节点。这种差异使得字典树在处理字符串时更具灵活性，能更自然地表示字符之间的关系。

- **路径具有明确语义**

二叉树的路径通常没有特定语义，主要用于搜索和排序操作。而字典树的每条路径都代表一个有意义的字符串，这使得字典树在字符串查找、前缀匹配等操作上效率极高。例如，要查找以“pre”为前缀的所有单词，只需在字典树中找到“pre”对应的路径，然后遍历该路径下的所有子树，就能找到所有符合条件的单词。

2.3.2 字典树的性质

前面我们已经对字典树的定义、核心特征，以及与二叉树的区别有了清晰的认识。接下来，我们将一起深入探究字典树在**时间和空间**层面展现出的独特性质，这将帮助大家更好地理解和运用这一数据结构。

时间复杂度特性

字典树在插入和查询操作上，有着出色的时间复杂度表现，均只需**O(|S|)**。这里的|S|，指的是要插入或查询的字符串的长度。

在**插入操作**中，当我们要向字典树中插入一个新字符串时，从根节点开始，按照字符串中字符的顺序，沿着对应的边依次向下遍历。如果某条边不存在，就创建一条新边。例如，向字典树中插入单词“banana”，从根节点出发，找到代表“b”的边，若该边存在则顺着它移动到下一个节点；若不存在，就创建一条新的“b”边。接着对“a”“n”等后续字符执行同样操作，直到整个单词插入完成。由于这个过程与字符串的长度成正比，因此插入操作的时间复杂度为O(|S|)。

查询操作的过程与插入类似。同样从根节点出发，根据待查询字符串的字符顺序，沿着相应的边进行遍历。因为查询过程同样只需要遍历字符串的每个字符一次，所以查询操作的时间复杂度也是O(|S|)。

空间特性

字典树在空间利用上非常**高效**，这主要得益于它能**让不同字符串共享相同的前缀**。如当字典树中同时存储“apple”和“app”这两个单词时，由于它们前三个字符“app”是相同的，在字典树里，这部分前缀只需要存储一次。这种共享前缀的特性，在存储大量具有相同前缀的字符串时，优势尤为明显。当在存储包含“program”前缀的众多单词，“program”这部分前缀只需存储一次，大大节省了存储空间。

然而，当字符串之间的公共前缀较少时，字典树的空间优势就会减弱，甚至可能会占用较多的空间。

2.3.3 字典树的实现

前面我们详细了解了字典树的定义、特性，现在就来动手实现字典树的基本功能，包括**初始化、插入和查询操作**。我们将使用C++ 语言来完成代码编写，通过具体的代码实现，加深对字典树的理解。

步骤1：定义存储结构

```
const int N = 1e5+10, M = 26; // N:最大结点数 M:字符集大小
int trie[N][M];   // trie[p][c]表示p结点通过字符c到达的结点编号
bool isEnd[N];     // 记录结点是否为单词结尾
int idx = 0;       // 当前可分配的结点编号（从1开始）
  return go(f, seed, [])
}
```

图2.3.3 定义存储结构

在这里，我们定义了一个二维数组 trie 来存储字典树的结构。trie[p][c] 表示从编号为 p 的节点，通过字符 c 所到达的下一个节点的编号。isEnd 是一个布尔类型的数组，用于标记某个节点是否是一个单词的结尾。idx 记录了当前可以分配的新节点的编号，初始时为 0，并且我们约定从 1 开始分配新节点。

步骤2：进行初始化

```
void init() {
    memset(trie[0], 0, sizeof(trie[0])); // 根结点编号为0
    idx = 1; // 下一个可用结点从1开始分配
}
```

图2.3.4 进行初始化

在 init 函数中，我们使用 memset 函数将 trie 数组中根节点（编号为 0）的所有子节点编号初始化为 0，表示根节点刚开始没有任何子节点。同时，将 idx 设置为 1，以便下一次创建新节点时从编号 1 开始。

插入操作实现

在 insert 函数中，我们从根节点（编号为 0）开始，遍历要插入的字符串 s 的每个字符。将字符转换为对应的数字（例如 'a' 转换为 0，'b' 转换为 1 等），检查当前节点 p 是否存在对应字符的子节点。如果不存在，就初始化一个新

节点，并将当前节点 p 的对应子节点编号设置为新节点的编号，同时更新 idx。然后移动到新的子节点继续处理下一个字符。当整个字符串处理完毕后，将最后一个节点标记为单词结尾。如图2.3.5的流程图所示，更加清晰的看到插入是如何操作的。

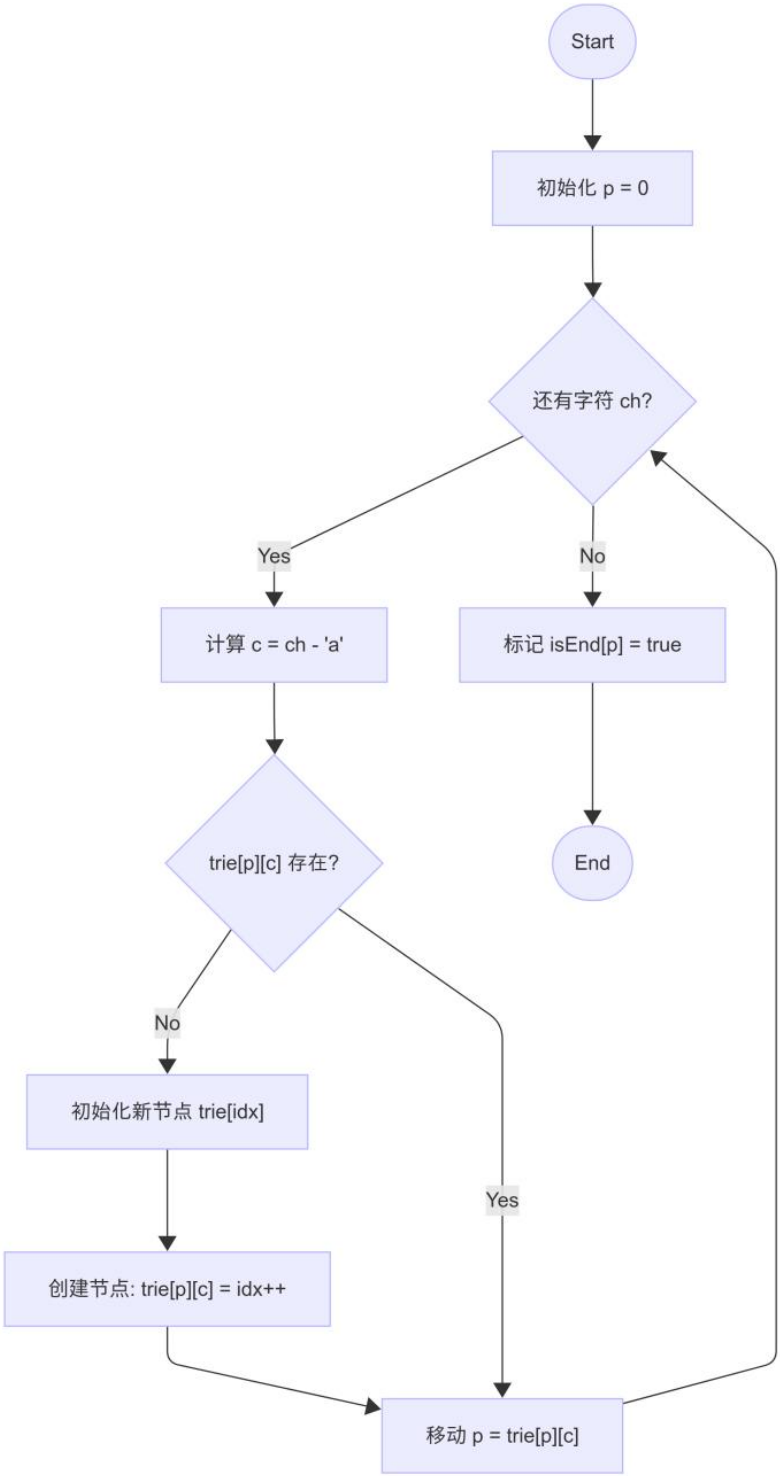


图2.3.5 插入操作流程图

插入操作是将一个字符串插入到字典树中，具体代码如下：

```
void insert(string s) {
    int p = 0; // 从根结点出发
    for(char ch : s) {
        int c = ch - 'a';          // 字符转数字
        if(!trie[p][c]) {          // 没有对应子结点
            memset(trie[idx], 0, sizeof(trie[idx])); // 初始化新结点
            trie[p][c] = idx++;    // 创建新结点
        }
        p = trie[p][c];            // 移动到子结点
    }
    isEnd[p] = true;               // 标记单词结尾
}
```

图2.3.6 插入操作代码

查询操作实现

在 search 函数中，同样从根节点开始，遍历要查询的字符串 s 的每个字符。将字符转换为数字后，检查当前节点 p 是否存在对应字符的子节点。如果不存在，直接返回 false，表示该字符串不存在于字典树中。如果能顺利遍历完整个字符串，最后检查最后一个节点是否被标记为单词结尾，以此来确定该字符串是否精确存在于字典树中。

查询操作是判断一个字符串是否存在于字典树中，代码如下：

```
bool search(string s) {
    int p = 0;
    for(char ch : s) {
        int c = ch - 'a';
        if(!trie[p][c]) return false;
        p = trie[p][c];
    }
    return isEnd[p]; // 必须精确匹配
}
```

图2.3.7 查询操作代码

通过以上代码的实现，我们初步完成了字典树的基本功能。大家可以在实际编程中运用这些代码，处理字符串相关的问题，进一步体会字典树的优势和特点。

2.3.4 字典树的应用

接下来我们来试着用所学到的知识解出下面的问题。

题目描述

某校长任命你为特派探员，每天记录他的点名。校长会提供化学竞赛学生的人数和名单，而你需要告诉校长他有没有点错名。

输入格式：

第一行一个整数 n，表示班上人数。接下来 n 行，每行一个字符串表示其名字（互不相同，且只含小写字母，长度不超过50）。第 n+2 行一个整数 m，表示教练报的名字个数。接下来 m 行，每行一个字符串表示教练报的名字（只含小写字母，且长度不超过 50）。

输出格式：

对于每个教练报的名字，输出一行。如果该名字正确且是第一次出现，输出 `OK`，如果该名字错误，输出 `WRONG`，如果该名字正确但不是第一次出现，输出 `REPEAT`。

步骤1：构建字典树

首先定义字典树的节点类，每个节点包含一个字典用于存储子节点（键为字符，值为子节点对象），以及一个布尔值表示该节点是否是一个单词的结尾。

遍历班上学生的名单，将每个学生的名字插入到字典树中。插入过程是从根节点开始，依次处理名字中的每个字符，如果当前字符对应的子节点不存在，则创建一个新的子节点，然后移动到该子节点继续处理下一个字符，直到名字的所有字符处理完，并将最后一个节点标记为单词的结尾。

步骤2：处理校长点名

对于校长报出的每个名字，从字典树的根节点开始进行查找。依次处理名字中的每个字符，检查当前字符对应的子节点是否存在。如果不存在，说明这个名字不在字典树中，即名字错误，输出 WRONG。如果当前字符对应的子节点存在，继续处理下一个字符，直到名字的所有字符处理完。此时检查最后一个节点

是否标记为单词的结尾，如果不是单词的结尾，也说明名字错误，输出 WRONG；如果是单词的结尾，说明名字是正确的。

为了判断名字是否是第一次出现，需要额外使用一个数据结构（如集合）来记录已经出现过的名字。当确定名字正确后，检查该名字是否在记录已出现名字的集合中。如果不在，输出 OK，并将该名字加入到集合中；如果在，说明该名字不是第一次出现，输出 REPEAT。

步骤3：重复上述步骤

对校长报出的所有名字，都重复上述查找和判断的过程，直到所有名字处理完毕。

通过以上步骤，利用字典树的特性来高效地存储和查找学生名字，从而实现对校长点名情况的准确判断。

核心代码展示

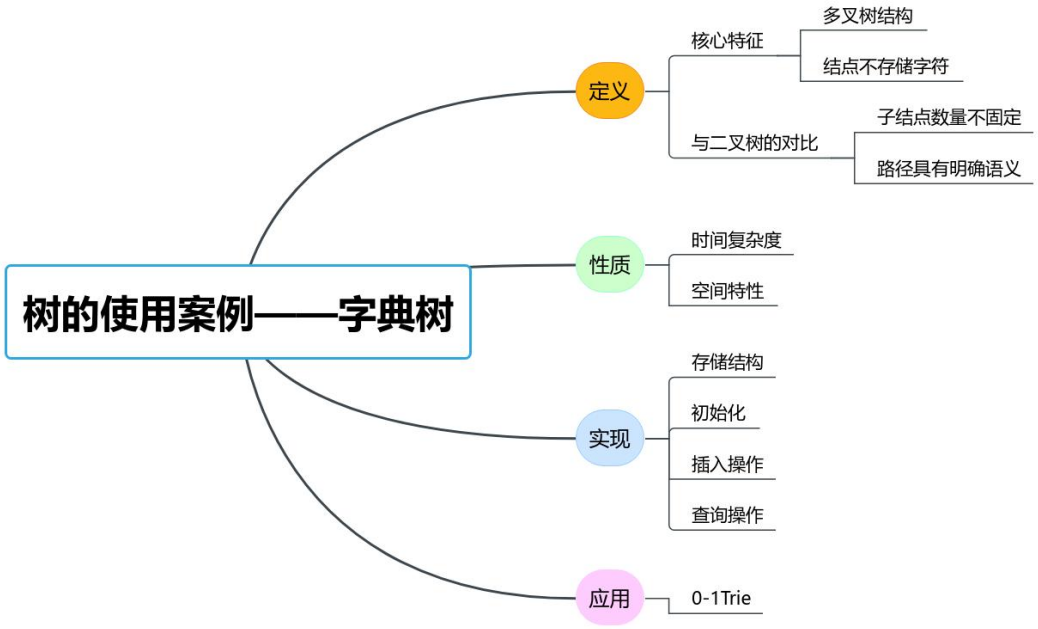
```
int cnt[N]; // 新增计数数组

void insert(string s) {
    int p = 0;
    for(char ch : s) {
        int c = ch - 'a';
        if(!trie[p][c]) trie[p][c] = idx++;
        p = trie[p][c];
        cnt[p]++; // 经过该结点的次数+1
    }
    isEnd[p] = true;
}
```

图2.3.8 统计前缀出现次数

拓展提升：0-1Trie

01-trie 是指字符集为 {0, 1} 的 trie。将数的二进制表示看做一个字符串，就可以建出01-trie。trie可以方便的统计在某一位上某种数字出现了多少次。因此，这个数据结构可以用于异或极值、异或和等与出现次数有关的问题。

知识图谱



练习提升

1. 给定一棵 n 个点的带权树，结点下标从 1 开始到 n。寻找树中找两个结点，求最长的异或路径。（注：异或路径指的是指两个结点之间唯一路径上的所有边权的异或。）

2. 给定一个字典树和一系列单词，编写一个程序来将这些单词插入到字典树中。假设字典树中的每个节点存储一个字符，并且每个节点有一个布尔值 isEnd 来标记单词的结尾。

• 具体要求：

2.1 编写一个 insert 函数，该函数接收一个字符串 s 作为参数。

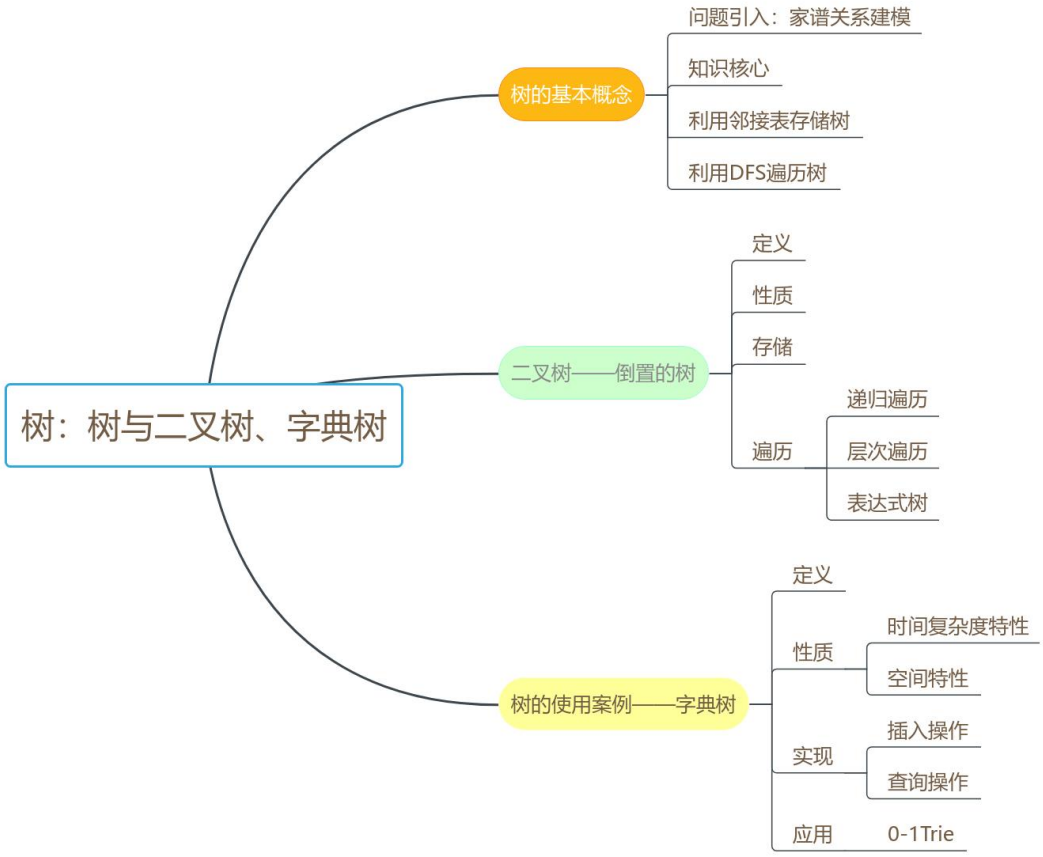
2.2 从字典树的根节点开始，逐个字符地检查字符串 s 中的字符。

2.3 如果字符对应的子节点不存在，则创建一个新的子节点，并将其链接到当前节点。

2.4 更新 isEnd 标记，以指示字符串 s 的结尾。

总结 评价

1. 下图展示了本章的核心概念与关键能力，请同学们对照图中的内容进行总结。



2. 根据自己的掌握情况填写下表。

学习内容	掌握程度
树的基本概念	□不了解 □了解 □理解
用代码实现树的遍历	□不了解 □了解 □理解
二叉树的定义和关键术语	□不了解 □了解 □理解
二叉树的遍历	□不了解 □了解 □理解
字典树的基本特征与实现	□不了解 □了解 □理解
字典树的应用	□不了解 □了解 □理解

小结

在本章中，我们围绕树这一核心主题展开了深入探索。通过剖析树形结构的层次关系与递归特性，我们不仅掌握了二叉树的前序、中序、后序遍历等基础操作，更通过竞赛真题理解了完全二叉树、满二叉树等特殊结构的应用场景。在字典树的专题中，我们通过字符串匹配、前缀统计等经典问题，领略了这种高效数据结构在信息检索中的独特优势。本章内容着重培养从问题特征出发选择数据结构的能力——面对需要体现层次关系、快速前缀匹配或递归求解的问题时，树结构往往能提供更优雅的解决方案。希望同学们能将二叉树的递归思维与字典树的空间优化技巧融会贯通，在面对搜索路径优化、字符串处理、最优决策树构建等复杂问题时，展现出更扎实的算法设计与问题拆解能力，为后续高阶数据结构的学习奠定坚实基础。